Introduction to Parallel Computing and Scientific Computation
Project: Particle Dynamics

by

Isabel Figueroa

1.0 How the Particle Dynamics program works

Particle Dynamics (PD) is a tool used to analyze the flow of granular material. This discrete method of simulation uses the Newton's laws of motion to describe the trajectories of individual particles. The time evolution of these trajectories determines the global flow of the granular material. Cohesive and adhesive systems can be studied by implementing the corresponding contact mechanics to calculate the contact force between particles. PD can also simulate heat transfer, as well as the liquid transfer in the granular media.

1.1 Physical Description

The equations that describe the particle motion are:
Linear Motion:

$$m_p \frac{dv_p}{dt} = -m_p g + F_n + F_t \tag{1-1}$$

Angular Motion:

$$I_p \frac{d\omega_p}{dt} = F_t \times r \tag{1-2}$$

where F_n and F_t are the inter particle forces – normal and tangential, respectively – acting on the particle and are functions of contact, drag, pressure and capillary interactions.

1.2 Numerical Methods

The Newton's equation of motion is solved by using a modification of the Gear predictor-corrector method. It involves two stages with a force evaluation in between. First, the new positions are calulated using

$$r(t + \Delta t) = r(t) + \Delta t v(t) + \frac{1}{2}\Delta t^2 a(t) \tag{1-3}$$

The velocities at mid-step are computed using

$$v(t + \frac{1}{2}\Delta t) = v(t) + \frac{1}{2}\Delta t a(t) \tag{1-4}$$

The force and accelerations at the time $t + \delta t$ are them computed, and the velocity move completed

$$v(t + \Delta t) = v(t + \frac{1}{2}\Delta t) + \frac{1}{2}\Delta t a(t + \Delta t) \tag{1-5}$$

The size of the time step depends on the type of contact mechanics used,

1.3 Modular structure of the program

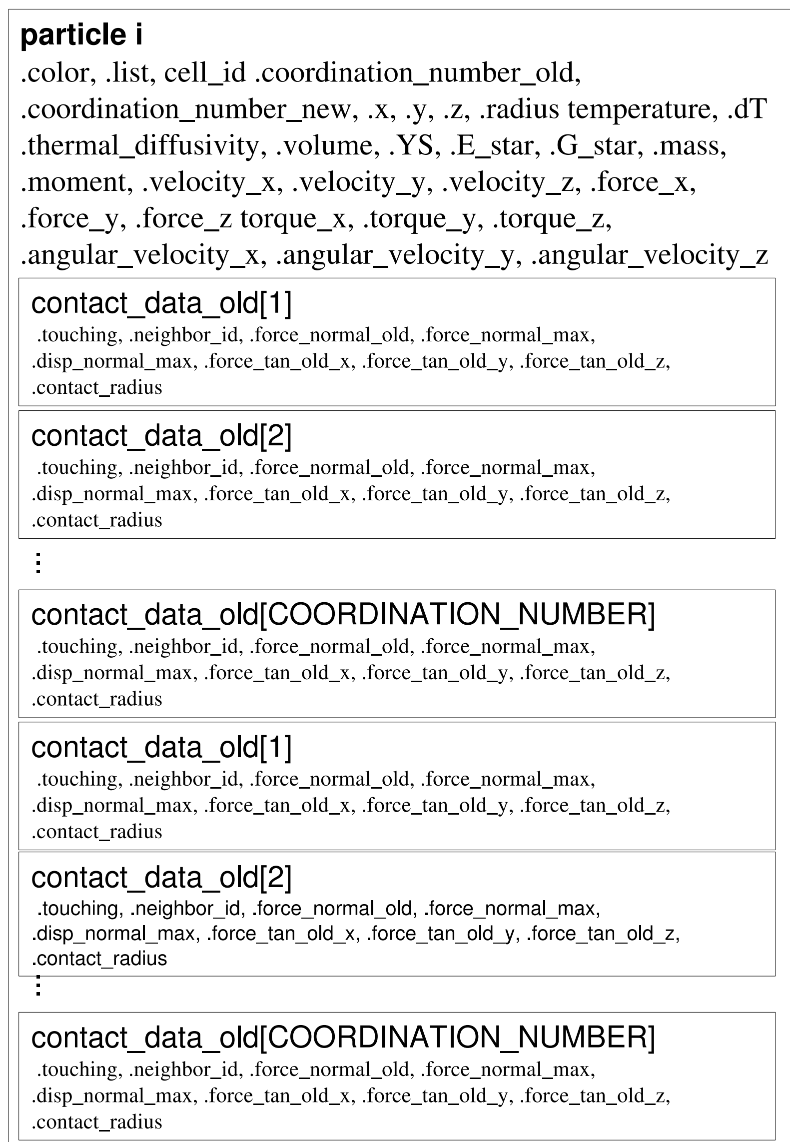
The PD has been written in several modules in c language. The program works in dimensionless variables. The input to this program are:

- An input file, containing the x, y, z coordinates of the particles, their initial velocity in the x, y, z direction, their initial angular velocity in the x, y, z direction, the particle radius, the particle initial temperature and the particle color.
- The program have to be given the total number of particles, the number of wall particles, the input file. It also allows to specify other option like the type of material, the type of simulations (static, tumbler, couette), heat transfer, etc. If these are nor specified the program just work with the default options.

The following section attempt to present how each of these interacts with the main program as well to describe what does each modules do. The data structures used in the program are also described in this section.

1.3.1 Structures used by the program

The program works with structure called Particle. In each of the fields of the structure different particle properties are stored. A number equal to the coordination number of 'Contact_Data' structure are nested to the 'Particle' structure. In each of the 'Contact_Data' fields the data related to the contact is stored. The following picture present how the information is managed in the program.

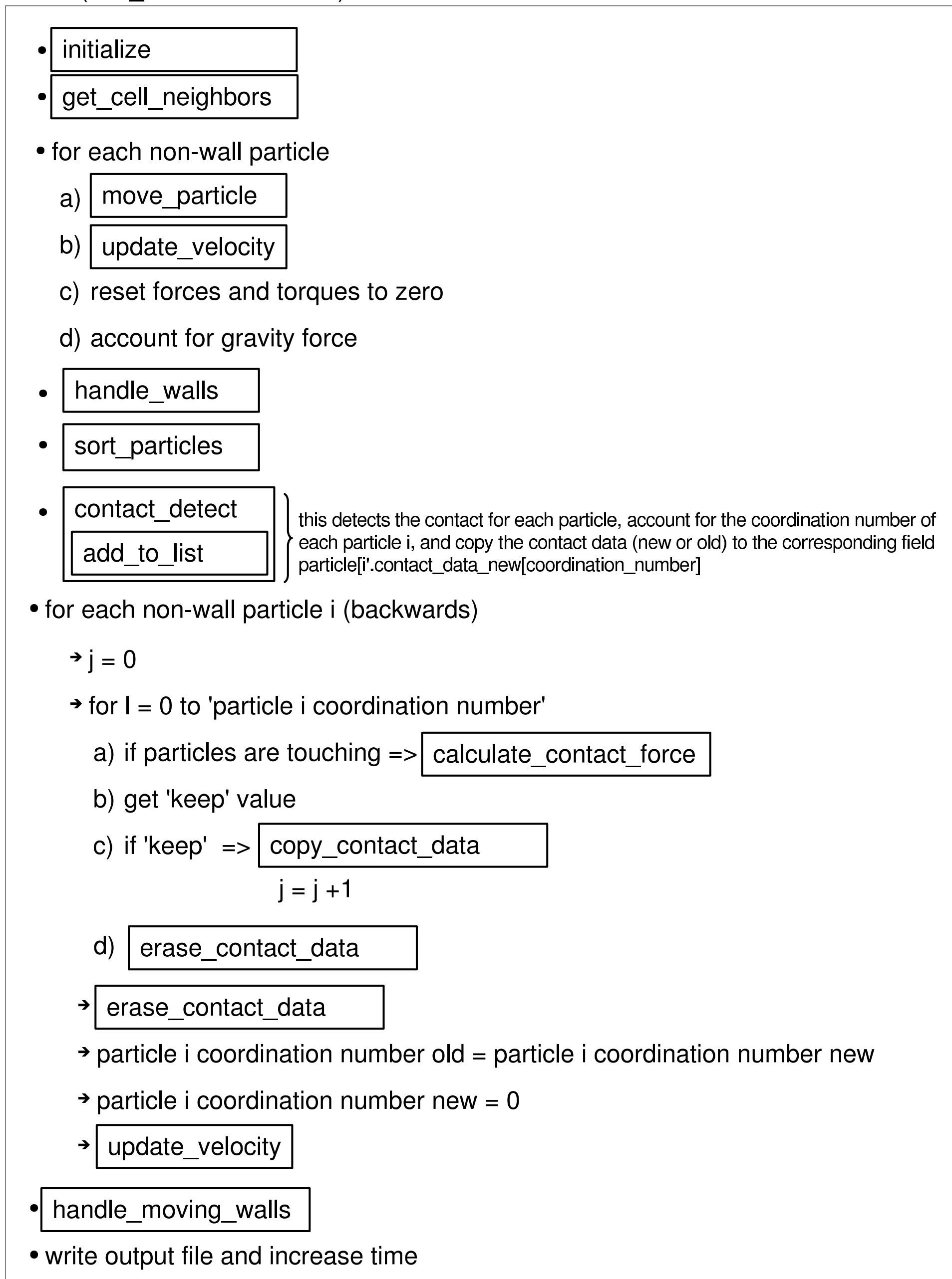


Structures used in PD.

Another array that PD uses is **cell[icell]** for icell=0...(number_of_cells-1). It saves the x, y and z index for the cell in .x .y and .z, the head of the cell in cell[icell].begin_of_cell, and the neighbor cell ids in cell[icell].neighbor[j] for j=0..cell_coordination_number.

1.3.2 Diagram

The main flow of the program start initializing all the variables by using initialize.c at the beginning of main.c. The advance in time is given by the execution of a while loop in main.c which will repeated until a terminating condition is satisfied. For each iteration in the while loop, and for each non-wall particle the program updates the particle position at the beginning of the time step. The particle velocity is updated for first time during the current time step, and the forces and torques are now reset to zero. The routine handle_walls now update the velocities and positions of the wall particles. Next, the contacts between particles are detected and the contact data is assigned to the new coordination numbers, this is done by contact_detect. The contact data between particles i and j are kept in the data contained by the particle of highest index. That explains why the next for loop go backwards. For each non-wall particle, and for each of the contact detected for this particle, the contact force between particles is calculated. A value (keep) is also assigned in calculate_contact_force.c. The data is erased or copied accordingly. The particle velocity is now updated for a second time in this current time step. Finally, the data will be written to a file every a determined quantity of time steps. The last part of the while loop checks for this writing condition and generates the output file. The main flow of the program is diagrammed is in the next figure.



PD flow diagram.

1.3.3 Module description

In this section, the main modules of the PD program are described respect to their main objective and where are used.

a) **initialize.c** This routine is run at the beginning of the program, and initialize all the particle[i] array, giving values to properties such as radius, mass, momentum,

mechanical properties, as well as their initial positions and velocities. It also resets all the forces and torques to zero

b) **get_cell_neighbors.c** This routine is run only once during one execution of the program. It is called by main.c and it gives the information cell[i].neighbor[j], for all the particles i and for j=0..cell_coordination_number.

c) **contact_detect.c** is called by main.c and detects the all the particles that may be in contact with the particle i. This routine calls add_to_list for each contact that has been detected.

d) **sort_particles.c** is called by main.c and this is executed once for every time step. This routine identifies in which cell the particle is in. In this process two arrays are generated

- The head of the list, cell[i].begin_of_list, this array assigns one element for each cell and corresponds to the particle number of one of the particles corresponding to that cell.
- The following particle in the list, particle[i].list, contains the number of the next molecule in the cell. If one follows the links of the particles in one cell, one eventually will find the element of the list which is -1. This indicates that there is no more particles in that cell.

e) **add_to_list.c** is called by contact_detect. This routine determines if the two particles i and j are in contact, the contact data will be stored the particle i structure (i¿j). The routine will look if this is and old or new contact. In case of old contacts, the routine will assign a new contact index and move the contact data to particle[i].contact_data[new_index]. For new contacts, old information is erased by using erase_contact_data.m

f) **copy_contact_data.c** : This routine copies the contact data whenever this should be kept.

g) **erase_contact_data.c** : This routine erases the contact data when the particle contact is finished.

h) **move_particle.c** update the particle position using the current particle velocity and acceleration. It is called in handle_walls.c and main.c

i) **update_velocity.c** : the velocity update is done twice during one time step, using the acceleration of the previous time step and the current one, respectively. This routine is called in handle_walls.c and main.c.

j) **calculate_contact_force.c** : this routine calculate the contact force according the contact mechanics that is used (cohesive, adhesive, etc). It returns a value (keep) that indicates if the contact data should be kept or not. It is called by main.c.

k) **handle_walls.c** : manages the wall particle movements.

2.0 Parallel Implementation of PD

The parallel version of this PD sofware is implemented by using MPI.

2.1 Structures

The main problem in the implementation of the parallel version of the code is how be build the structures in MPI, in order to be able to pass them between processors. Some of them are dynamically allocates; therefore, it is important to set each component address correctly, and account for the relative displacement in each case. The code to accomplish that is presented in this secction.

```
MPI_Datatype Celltype;
 MPI_Datatype type_cell[3]={MPI_INT, MPI_INT, MPI_INT};
 int blocklen_cell[3]={CELL_COORD_NUMBER,1,1};
 MPI_Aint disp_cell[3];
 MPI_Address(&(cell[0].neighbors), &disp_cell[0]);
 MPI_Address(&(cell[0].begin_of_list), &disp_cell[1]); MPI_Address(&
(cell[0].next_in_node), &disp_cell[2]); disp_cell[2] -= disp_cell[0];
disp_cell[1] -= disp_cell[0]; disp_cell[0] -= disp_cell[0];
MPI_Type_struct(3, blocklen_cell, disp_cell, type_cell, &Celltype);
MPI_Type_commit(&Celltype);
```

Node structure.

```
MPI_Datatype Celltype;
 MPI_Datatype type_cell[3]={MPI_INT, MPI_INT, MPI_INT};
 int blocklen_cell[3]={CELL_COORD_NUMBER,1,1};
 MPI_Aint disp_cell[3];
 MPI_Address(&(cell[0].neighbors), &disp_cell[0]);
 MPI_Address(&(cell[0].begin_of_list), &disp_cell[1]); MPI_Address(&
(cell[0].next_in_node), &disp_cell[2]); disp_cell[2] -= disp_cell[0];
disp_cell[1] -= disp_cell[0]; disp_cell[0] -= disp_cell[0];
MPI_Type_struct(3, blocklen_cell, disp_cell, type_cell, &Celltype);
MPI_Type_commit(&Celltype);
```

Cell structure.

```
MPI_Datatype Particletype;
MPI_Datatype type[30]={MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,
        MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,
        MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,
        MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,MPI_INT,MPI_INT,MPI_INT,
        MPI_DOUBLE,MPI_DOUBLE,
        MPI_DOUBLE,MPI_DOUBLE,MPI_DOUBLE,
        MPI_INT};
int blocklen[30]={1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1,
    COORDINATION_NUMBER, COORDINATION_NUMBER,
    COORDINATION_NUMBER, COORDINATION_NUMBER, COORDINATION_NUMBER,
    COORDINATION_NUMBER};
MPI_Aint disp[30];
MPI_Address(&(particle[0].x), &disp[0]);
MPI_Address(&(particle[0].y), &disp[1]);
MPI_Address(&(particle[0].z), &disp[2]);
MPI_Address(&(particle[0].radius), &disp[3]);
MPI_Address(&(particle[0].E_star), &disp[4]);
MPI_Address(&(particle[0].G_star), &disp[5]);
MPI_Address(&(particle[0].YS), &disp[6]);
MPI_Address(&(particle[0].mass), &disp[7]);
MPI_Address(&(particle[0].moment), &disp[8]);
MPI_Address(&(particle[0].velocity_x), &disp[9]);
MPI_Address(&(particle[0].velocity_y), &disp[10]);
MPI_Address(&(particle[0].velocity_z), &disp[11]);
MPI_Address(&(particle[0].force_x), &disp[12]);
MPI_Address(&(particle[0].force_y), &disp[13]);
MPI_Address(&(particle[0].force_z), &disp[14]);
MPI_Address(&(particle[0].torque_x), &disp[15]);
MPI_Address(&(particle[0].torque_y), &disp[16]);
MPI_Address(&(particle[0].torque_z), &disp[17]);
MPI_Address(&(particle[0].angular_velocity_x), &disp[18]);
MPI_Address(&(particle[0].angular_velocity_y), &disp[19]);
MPI_Address(&(particle[0].angular_velocity_z), &disp[20]);
MPI_Address(&(particle[0].number_of_neighbors), &disp[21]);
MPI_Address(&(particle[0].old_neighbors), &disp[22]);
MPI_Address(&(particle[0].list), &disp[23]);
MPI_Address(&(particle[0].disp_normal_max[0]), &disp[24]);
MPI_Address(&(particle[0].force_normal_max[0]), &disp[25]);
MPI_Address(&(particle[0].force_tan_old_x[0]), &disp[26]);
MPI_Address(&(particle[0].force_tan_old_y[0]), &disp[27]);
MPI_Address(&(particle[0].force_tan_old_z[0]), &disp[28]);
MPI_Address(&(particle[0].index[0]), &disp[29]);
for (i=29; i>0; i--) disp[i] -= disp[0];
disp[0] = 0;

MPI_Type_struct(30, blocklen, disp, type, &Particletype);
MPI_Type_commit(&Particletype);
```

Particle structure.

2.1.1 How the job is divided

In order to make this code parallel is necessary to assign different jobs to each of the processors. The sequential code divides the spaces into fixed cells. Each node has assign certain number of cells. In the case of two nodes, the space is divided with a vertical line through the middle of the tumbler. In the node structure, the head of the list of the linked list is saved.

The most computationally expensive part of the program is the routine to calculate the force between the particles, so this work is the part that needs to be divided into the two

processor. The particle structure containing the contact data is combined and, then, the particle positions are updated in one processor. These procedure can be summarized in the following

1. Particle initialization.

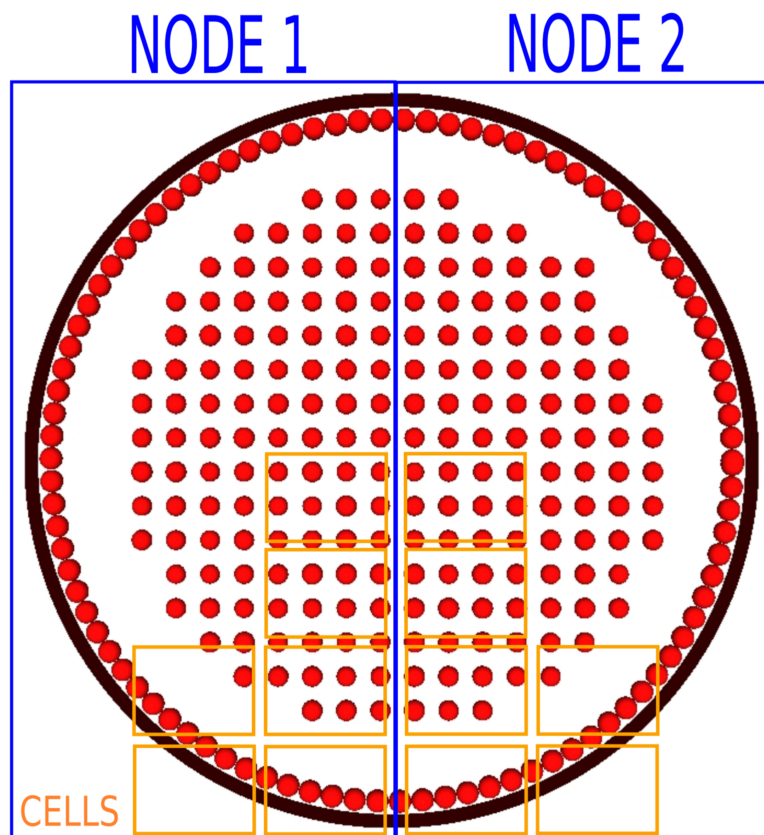
2. Structure construction in MPI

3. Send structure to all processors: *Cell*, *Node*,*Number*,*Size*, and *Particle*.

4. Initialize

5. Loop

    - Send the updated Particle structure from the main processor to all the rest
    - Calculate the forces in each processor
    - Send the updated Particle structure from all processors to main processor



How the particles are assigned for each node.

3.0 How to run PD

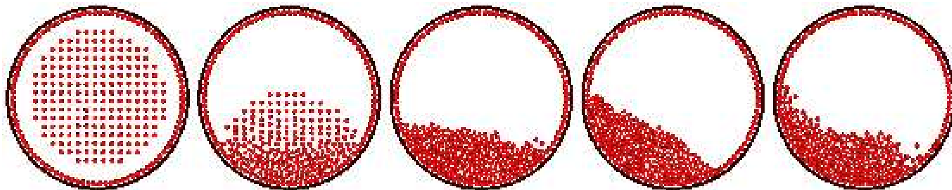
3.0.2 How the job is divided

In order to run PD a file containing the initial conditions is needed. In this file, the initial positions, velocities, angular velocities for all the particles.

The input arguments that are required are:

- number of particles
- number of wall particles
- contact mechanics (plastic, elastic, cohesive, adhesive)
- initial file name

The program generates files indicating the position of the particle as the time passes. This files can be used to generate pictures, and therefore movies of the progress of the granular material.



Example of PD results.